J. Symbolic Computation (1991) **11**, 349-368

A Rewrite-based Type Discipline for a Subset of Computer Algebra

H. COMON, D. LUGIEZ AND PH. SCHNOEBELEN†

Laboratoire d'Informatique Fondamentale et d'Intelligence Artificielle, Grenoble, France

(*Received* 2 *February* 1988)

This paper is concerned with the type structure of a system including polymorphism, type properties and subtypes. This type system originates from computer algebra but it is not intended to be the solution of all type problems in this area.

Types (or sets of types) are denoted by terms in some order-sorted algebra.‡ We consider a rewrite relation in this algebra, which is intended to express subtyping. The relations between the semantics and the axiomatization are investigated. It is shown that the problem of type inference is undecidable but a narrowing strategy for semi-decision procedures is described and studied.

Introduction

This paper deals with a type system that includes polymorphism, type properties, and subtypes. This type system—relevant in (a subset of) computer algebra which will be our case study throughout this paper—aims at two contradictory goals which must be solved *simultaneously*:

1. static type checking; and
2. the possibility to omit typing information.

Static type checking allows the detection of type errors, at least most of them, at compile time, and this helps in writing correct programs. Static type checking also allows the generation of efficient code because it avoids the run-time overhead coming from dynamic type-checking. On the other hand, type systems with dynamic checking usually have a much richer structure, and computer algebra is a domain in which such rich type structures are required. Actually, most systems have chosen a dynamic type calculus (e.g. Macsyma (1984), but see also Abdali *et al.*, 1986) and indeed, static type-checking is undecidable, even in a restricted case, as proved in this paper.

The second requirement contradicts the first one, since it allows the user to have incompletely typed expressions, but it is of great importance in computer algebra because it is extremely tedious and error prone to give explicitly the type of each algebraic expression and all its subexpressions as shown by the example in section 1.1.

A solution is to use a *type inference* algorithm, which allows the system effectively to compute statically the type of every (sub-)expression with no (or only a few) typing information provided by the user. In a sense, every type-checking system does some type

† Order irrelevant. Authors current addresses: H. Comon, LRI, Bât 490, Univ. Paris Sud, 91405 Orsay Cedex, France, D. Lugiez, CRIN, Campus Scientifique, 615 r. Jardin Botanique, BP 101, 54600 Villers-Les-Nancy, France, Ph. Schnoebelen, LIFIA-IMAG, 46 Av. Félix Viallet, 38033 Grenoble Cedex, France.

‡ A similar framework is described in Smolka (1989). The present paper was submitted in February 1988; we had not enough information at this time for an accurate comparison with Smolka's work.

0747-7171/91/040349+20 $03.00/0

inference, but in order to free the user from giving any type information, the type system must be very simple or very regular (as in ML, Gordon *et al.*, 1979). The problem with computer algebra is that the type system is already fixed by the application, it is not simple and not so regular.

In computer algebra systems of the first generation such as Reduce (Hearn, 1984) or Macsyma (1984), the problem has been "easily" solved because these systems do not implement the key notions of *polymorphism*, *properties* (in the sense of Futatsugi *et al.* (1985)) and *subtypes* (as in Smolka (1989)), or only in a very restricted way. Polymorphism and subtypes are basic features of the new systems (Fortenbacher *et al.*, 1985; Abdali *et al.*, 1986) since these systems are intended to be used in a larger area than computer algebra, which implies that they rely on safe and extensible foundations.

Subtypes and polymorphism are very natural, and as shown in section 1 even simple algebraic expressions involve them. But with them the type inference problem becomes very complex.

Computer algebra systems thus far have dealt with type inference in a very ad hoc way since either they prune the polymorphism or they require some explicit typing for non-standard expressions. In other languages, type inference with some polymorphism does exist, like in ML or OBJ2 (Futatsugi *et al.*, 1985), but it is still restricted. In ML, type inference is performed by a simple algorithm which relies on unification (Milner, 1978), but subtypes are not allowed. In OBJ2, polymorphism is achieved and some subtyping is allowed but it is limited in each module to a finite ordered sequence of subtypes. In Scratchpad (Jenks & Sutor, 1987), a uniform approach, based on categories, addresses all three aspects, but it requires some heuristics—specially when typing expressions—to be effective.

Presently, a clean and transparent description of a type inference mechanism in computer algebra (including correctness proofs) is lacking. This paper aims at providing such a description:

Syntax: (polymorphic) types are terms in an order-sorted algebra where sorts denote properties.

Semantics: we use a set-theoretic semantics of types.† Polymorphic types denote the intersection of their semantic instances.

Subtyping: we define a syntactic relation $\rightarrow$ (called "derivation") and a semantic subtyping relation $\lhd$. We study the adequation of $\rightarrow$ with respect to $\lhd$.

Type Inference: we propose a type inference mechanism leading to "most general" types of functional expressions.

This type system provides a clear and clean description of typing, and is relevant as far as a regular enough subset of computer algebra is concerned. In section 1 we illustrate by means of examples what kind of problems appear in computer algebra and what kind of relationships we would like to express, then we propose our formalism in section 2. The main idea is to use rewrite rules to define $\rightarrow$: this gives a very powerful tool for describing the complex relationships that exist between computer algebra domains. Furthermore, from an operational point of view, the subtyping relation enjoys many properties of rewrite systems and the typing of an expression is similar to the problem of finding the types common to two given types, which is a unification problem. Unfortunately, one result of this paper is that even in a simplified model, the problem remains undecidable, which shows that some strong restrictions should be put on the subtyping

† Of course, this is not sufficient for the full description of algebraic structures.

relation in order to get a type inference algorithm that needs absolutely no user-given type annotation. However, in section 3, we show that when some not unreasonable restrictions are made it is possible to get an efficient (and complete) semi-decision procedure, close to the narrowing process (Hullot, 1980) for linear terms, which may be used for the subtype relation.

In this paper, we emphasize semantics. We believe it is important when describing a type system to give it some (natural) semantics because, in our opinion, typing mechanisms must be proven sound and, possibly, complete w.r.t. the semantics of types one adopts. This is even more crucial when these typing mechanisms are used automatically by the system without any user intervening, e.g. in type inference procedures.

The reader should keep in mind that this is an exploratory work where many interesting problems have not yet been investigated.

1. A Case Study

In this section, we first consider a typical example of type inference in computer algebra and analyse it in detail, emphasizing the requirements we have to make about the type system, and trying to pinpoint which kind of typing mechanism is used. Then, with this motivation, we begin to describe, still informally, which formalism can be used to describe this mechanism and to deal with these requirements. We shall then have a sufficiently clear idea of how we want to formalize our problems, which we do in section 2.

1.1. AN EXAMPLE OF TYPE ANALYSIS IN COMPUTER ALGEBRA

Let us consider the expression:

$$2/3+3*X$$

and suppose that it has been entered by the user of a computer algebra system. We may then ask which type should be given to such an expression. Obviously, this expression is a polynomial over the rational numbers. But how should this be deduced by the system? Note that several systems exist (Macsyma, 1984; Hearn, 1984) which are all able to deduce this fact. The problem is that their approach is not general† enough to allow them to deal with more complicated examples. In a sense, they have an ad hoc knowledge of some basic types commonly used in typical applications (e.g. integers, rationals, polynomials and matrices) but it is not possible to extend this knowledge in a clean and smooth way.

Given the previous expression, the system will parse it and treat it as a tree, as depicted in Figure 1.

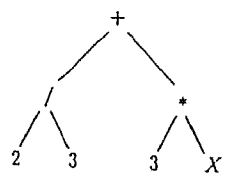

Figure 1. Parse Tree.

† The exception to this is Scratchpad which has a general typing mechanism.

The leaves are known to have type *INT* (for "integers") and *SYMBOL*. Let us first examine the left subtree, i.e. the expression 2/3. 2 and 3 are *INT*s and we should look for a way to apply the / operation to integers. In usual systems, the symbol / is *overloaded*: it may be used to divide integers, rational numbers, polynomials, ..., and, as integers are rationals which are themselves polynomials, all this can be done somewhat transparently, without precisely telling the user that / is overloaded. But this does not provide safe foundations: / denotes the *polymorphic* division operation, it is defined over any set having the structure of a field, and only there.

We should now try to use this polymorphic meaning of / to find the type of 2/3. We first note that *INT* is not a field, and indeed 2/3 is not an *INT*, but rather a *RAT* (i.e. a rational number). Since *INT* is a *subtype* of *RAT*, i.e., each element of *INT* also belongs—up to isomorphism—to *RAT*, 2 and 3 could also be seen as elements of *RAT*, which is a field. Therefore, we may infer a type for the expression by first embedding 2 and 3 into *RAT*. This example already demonstrates that our formalism will have to take into account both *subtyping* and *polymorphism*.†

Subtyping arises because any integer is also known to be a rational number, and we may say that *INT* is a subtype of *RAT*, which we write:

$$INT \lhd RAT. \tag{1}$$

In this introductory section, we need not be more precise about the semantics of subtypes: as computer algebra mainly deals with well known and widely used mathematical objects, the reader will always understand the examples we consider.

Polymorphism arises with our notion of division: informally, polymorphism is a concept which appears when a given operation or construction may be applied to a wide variety of different objects, often with the condition that they share a common (sub)structure. In our example, the polymorphic object is the division operation‡ and it is defined over *any field*, which we could write:

$$\forall t{:}Field,\ /{:}(t, t) \to t. \tag{2}$$

Here t is a type variable, as in ML, but it ranges only over types which are fields. So "being a field" is a *property* satisfied by some types, and which may be *required* by some polymorphic constructs. We shall use the intuitively clear notation "$\forall t{:}Field\ldots$" without further explanations until section 2.

Looking at the right subtree, i.e. the expression $3 * X$, we have to combine an *INT* and a *SYMBOL*. $*$ is polymorphic but it only requires that its arguments belong to (say) a Ring. X may be considered as a univariate polynomial with integer coefficients and with X as indeterminate, a type we write $UP[INT, X]$. *UP* is an example of a polymorphic type: it is parameterized by other types. Similarly, 3 may also be considered as a $UP[INT, X]$. This is possible because we clearly have:

$$INT \lhd UP[INT, X]. \tag{3}$$

In fact, the general result is rather:

$$\forall t{:}Ring,\ t \lhd UP[t, X]. \tag{4}$$

† Actually, we only consider *first-order* polymorphism.

‡ Different from *div*, the Euclidian division defined over Euclidian rings. Also, we ignore the problem of dividing by zero.

$t \lhd UP[t, X]$ is a knowledge about polynomials, of the kind we want to handle, while $INT \lhd UP[INT, X]$ is an ad hoc knowledge about polynomials with integer coefficients. Similarly, the rule $INT \lhd RAT$ we used earlier is but a special case of a more general result: any integral domain is a subtype of its fraction field, which we write:

$$\forall t: IntegralDomain,\ t \lhd FF[t]. \tag{5}$$

Once $2/3$ and $3 * X$ have been typed as a RAT and a $UP[INT, X]$ respectively, what is the type of the global expression $2/3 + 3 * X$? Again, these types have to be embedded into a (common) supertype. In this case we first have $RAT \lhd UP[RAT, X]$ as an instance of (4). Then we also have $UP[INT, X] \lhd UP[RAT, X]$, which is a consequence of rule (1) and the monotonicity of UP (this remark about monotonicity will be developed later). As a result, our expression has type $UP[FF[INT], X]$.

By a similar argument, we could have given it the type $FF[UP[INT], X]]$ (i.e. the fractions of polynomials with integer coefficients), but it is natural to choose $UP[FF[INT], X]$ because $UP[FF[INT], X] \lhd FF[UP[INT, X]]$, which is a special case of the general fact:

$$\forall t: IntegralDomain,\ UP[FF[t], X] \lhd FF[UP[t, X]]. \tag{6}$$

Saying that the expression is a polynomial over the rationals will always allow us to deduce, if need be, that it is also a ratio of polynomial with integer coefficients, while the converse is not true. In order to commit ourselves only when it is necessary, we shall always try to choose a *most general*† possible type.

1.2. EXPRESSING SUBTYPING INFORMATION

The example we have considered demonstrates that the type analysis performed in an ad hoc way in most current systems can be very nicely expressed using polymorphism and subtypes, in a way which is much closer to the real semantics of the mathematical objects we handle. Unfortunately, though these two notions may combine smoothly (see e.g. Cardelli & Wegner, 1985), they make it difficult to completely get rid of user-provided type information. Anyway we shall propose and investigate a formalism with which it is possible to describe a type system embedding a significant part of algebra.

Intuitively, a basic idea to handle polymorphism is to denote types by terms where a given set of function names (functors) denotes the construction of more complex types from simpler ones (e.g. UP, FF, ...) and where variables range over types. With this, our example suggests that the subtyping relation can be specified with universally quantified relations (called rules). A rule like $UP[FF[t], X] \lhd FF[UP[t, X]]$ denotes all the subtyping information that can be obtained by instantiating its variables in any possible way. Clearly, all the subtyping relations we used can be expressed in such a way.

We shall also have to assume some additional structure over the relation, namely that it has a so-called "replacement property": if T is a term containing some type t, what we write $T = T[t]$, then replacing t by a subtype yields a subtype of T:

$$t' \lhd t \Rightarrow T[t'] \lhd T[t].$$

This is very natural, and we just used this to deduce $UP[INT, X] \lhd UP[RAT, X]$ from $INT \lhd RAT$. From a semantical point of view, this replacement rule amounts to saying

† Note that this notion of "more general" type is not based on instantiation of variables.

that the type constructors are monotonic. On the other hand, the monotonocity requirement will prevent us from describing all subtyping relations, which is the price to pay to get a manageable system.

This formalism is very reminiscent of term rewriting systems, the only difference is that we have no requirements about the presence of variables of any side in the other one, and indeed we may define a reduction relation over types, written $\rightarrow$, defined by some basic rules like (5) and (6), which describe our knowledge of the basic mathematical structures, and by closing it under replacements and substitutions. This closure gives a new relation over terms, that we write $\rightsquigarrow$. Our intention is that the (semantic) subtyping relation, written $\lhd$, be expressed by the (syntactic) $\rightsquigarrow$ relation, i.e.:

$$t \lhd t' \Leftrightarrow t \rightsquigarrow t'.$$

By definition, $\rightsquigarrow$ is both reflexive and transitive. It is not necessarily antisymmetric as it is possible to have "cycles", a very natural example being based on the idempotence of FF: from $FF[FF[t]] \rightarrow FF[t]$ and rule (5), we may derive $FF[t] \rightsquigarrow FF[FF[t]] \rightsquigarrow FF[t]$. Thus $\rightsquigarrow$ is only a quasi-ordering, which could be used to induce an equivalence relation on types.

1.3. HANDLING PROPERTIES

A last problem is to handle the notion of *property* required by polymorphic constructs. As we have seen it with UP, a type constructor cannot meaningfully be applied over every type. $UP[R, X]$ is defined only if R is a ring (and then it denotes another ring). In a sense, the types are typed: some types are rings, some are fields, those which are fields are also rings, and so on.

Of course, a type (say INT) is not a ring in itself: it is a ring with operations 0, 1, + and *. Rather than parameterizing properties with function names,† it is possible to simply fix the names once and for all. Then a "*Ring*" is a ring with functions named 0, 1, + and * (most systems do the same thing (Abdali *et al.*, 1986)). It will then be possible to have several kinds of rings.

As, in practice, only a finite number of properties are involved in the type structures handled by computer algebra, i.e. one does not usually construct new properties from previous ones, a possible and sensible framework for integrating properties in type inference is *many-sorted algebra*; and since there clearly is a notion of inheritance between properties (e.g. any Field is a Ring), we use *order-sorted* (instead of just many-sorted) algebra (Goguen *et al.*, 1984; Goguen & Meseguer, 1987*b*).

We consider a fixed finite set of sorts (our properties) and a partial ordering over them, such that a term (i.e. a type) which is of sort *Field* is also known to have sort *Ring*. Now, the properties required by the type constructors may be denoted by giving them an order-sorted arity, like in:

$$FF : IntegralDomain \rightarrow Field.$$

Such order-sorted arities allow us to express that the polynomials over a ring form a ring, over a commutative ring form a commutative ring, over a field form an Euclidean ring, ...

Similarly, the polymorphic functions defined over polymorphic types may require that some properties be satisfied by the type of the argument: division is only defined over fields, which is expressed by (2).

† As it is done in LPG (Bert & Echahed, 1986) and OBJ2 (Futatsugi *et al.*, 1985).

Our problems and motivations are now sufficiently clear, and we may introduce our formalism without any further delay.

2. A Formalism for Polymorphic Subtypes with Properties

This section formalizes the ideas suggested by the case study of section 1. It presents a type system into which polymorphism with properties may be expressed, and where subtyping information is given through rewrite rules. This approach includes what Cardelli (1984) calls horizontal polymorphism (mechanism of instantiation) and vertical polymorphism (subtyping). Moreover, the use of many-sorted algebra to incorporate the notion of property, and of order-sorted algebra to allow inheritance among properties, is a new feature of our formalism.

We begin by defining a language for type presentations, and the derivation relation which is induced. Then a natural semantics is given to such presentations, where subtyping is defined as set inclusion. Finally, a couple of theorems relate the syntactic notion of derivation and the semantic notion of subtyping.

We shall not pay much attention to expressions themselves, only to their types. When required, we shall suppose that there is a language $\mathscr{L}$ of all the expressions of the computer algebra system, but we shall not define it.

2.1. ORDER-SORTED ALGEBRA

We first briefly recall the basic notions about order-sorted algebra that will be used in the paper, but we assume that the reader already knows about terms, occurrences (positions in terms) and substitutions. For more details about terms, the reader may refer to Huet & Oppen (1980), and to Goguen *et al.* (1984), Goguen & Meseguer (1987*a*) and Schmidt-Schauss (1987) for order-sorted algebra.

DEFINITION 1. An *order-sorted signature* is a tuple $(S, \leq, \Sigma)$ where:
S is a finite set of *sorts*.
$\leq$ is a partial ordering over S.
Σ is a finite $S^* \times S$-indexed family of finite sets of *function names*.

If $f \in \Sigma_{s_1 \times \cdots \times s_n, s}$, we say that f has *profile* $s_1 \times \cdots \times s_n \to s$ and *arity* n. A same function name f may have different profiles if they all have the same arity, so that we may speak of "the arity of f". A tuple $s_1 \times \cdots \times s_n$ is a *word* of length n. The ordering $\leq$ is extended to words by $s_1 \times \cdots \times s_n \leq s'_1 \times \cdots \times s'_n$ iff $s_i \leq s'_i$ for $i = 1, \ldots, n$. We use $|w|$ to denote the length of w, and λ to denote the empty word. A member of some $\Sigma_{\lambda, s}$ is the name of a *constant* and it has arity 0.

DEFINITION 2. The *initial term algebra* $T_{S,\Sigma}$ is the union of the sets T_s for $s \in S$, recursively defined by
$s_1, s_2 \in S,\ s_1 \leq s_2,\ t \in T_{s_1} \Rightarrow t \in T_{s_2}$.
$f \in \Sigma_{s_1 \times \cdots \times s_n, s},\ \forall i = 1, \ldots, n,\ t_i \in T_{s_i} \Rightarrow f(t_1, \ldots, t_n) \in T_s$.

It is sometimes useful to consider $T_{S,\Sigma}$ as a subset of the *unsorted* term algebra T_Σ: we say that $T_{S,\Sigma}$ contains the *well-sorted* terms of T_Σ.

When $X = \bigcup_{s \in S} X_s$ is an S-indexed family of countably infinite sets of variable names, disjoint from the function names, we may form the *free term algebra over X*, written $T_{S,\Sigma}(X)$, by considering every variable name $x \in X_s$ as a constant of sort s.

A *linear* term is a term in which no variable occurs more than once and a *ground* term is a term with no variables. The subterm of t at occurrence m is denoted t/m. The term obtained by replacing the subterm t/m of t by a term v is denoted $t[m \leftarrow v]$, and this grafting operation is only defined when the resulting term is well-sorted.

We shall often refer to "grounding substitutions". If one wants to be precise, a grounding substitution is "grounding" w.r.t. a set of variables. As it is cumbersome to always explicit the set of variables w.r.t. which a given substitution is grounding, we shall omit this indication: it will always be the set of variables of the term (the pair of terms, the substitution, ...) to which the substitution is applied.

DEFINITION 3. A signature $(S, \preceq, \Sigma)$ is *regular* if for all $f \in \Sigma_{w_1,s_1} \cap \Sigma_{w_2,s_2}$ s.t. $|w_1| = |w_2|$, we have:

$$\forall w \preceq w_1,\ w \preceq w_2, \exists w',\ s',\ w \preceq w',\ s' \preceq s_1,\ s' \preceq s_2, f \in \Sigma_{w',s'}.$$

Given a signature, it is very easy to check that this regularity condition is satisfied. Its purpose is to ensure that every term $t \in T_{S,\Sigma}$ has a *least sort*, written $sort(t)$. The definition easily extends when variables are considered. If $s = sort(t)$ we shall often write $t{:}s$ instead of just t when this makes things clearer. In the following, we only consider regular signatures.

DEFINITION 4. A substitution σ is *well-sorted* if $\forall x \in X_s$, $\sigma(x) \in T_s(X)$.

This simply says that it is allowed to substitute a variable of sort *Ring* by a term of sort *Field* but not the other way around. In fact, well-sorted substitutions are the natural morphisms of $T_{S,\Sigma}(X)$ while other substitutions are just meaningless syntactic objects. If σ and θ are well-sorted substitutions and t is a well-sorted term, then $\sigma.\theta$ and $\sigma(t)$ are well-sorted. We shall use **WSSub** to denote the set of all well-sorted substitutions.

2.2. THE LANGUAGE OF TYPES

Our type language is defined through a *presentation*, which is an order-sorted signature $(S, \preceq, \Sigma \cup X)$ together with a finite set of subtype rules $(l_i \to r_i)_{i=1,\ldots,m}$ where $l_i, r_i \in T_{S,\Sigma}(X)$. The variables in a rule are universally quantified, which explains why we used a notation such as $\forall t{:}$*IntegralDomain*, $t \lhd FF[t]$ for rule (5) in section 1.1.

DEFINITION 5. Given a presentation $\mathbf{P} = (S, \preceq, \Sigma \cup X, (l_i \to r_i)_i)$, a *type* is a well-sorted term of the order-sorted algebra $T_{S,\Sigma}(X)$ where:

$(S, \preceq, \Sigma)$ is a regular signature such that $\forall s \in S$, $T_s \neq \emptyset$.

The sorts are called properties and have, e.g., Field, Ring, ..., as typical elements.

The ordering $\preceq$ reflects the strength of the properties. For instance, Field $\prec$ Ring.

The operator names of Σ are called type constructors, and have, e.g., *FF*, ..., as typical element.

A type denoted by a ground term is called a *ground type*, a non-ground type is called a *polymorphic type*.

2.3. DERIVED TYPES

The rules contained in a presentation $\mathbf{P}=(S, \preceq, \Sigma, (l_i \to r_i)_i)$ are used to derive types by "rewriting".†

DEFINITION 6. The derivation relation over $T_{S,\Sigma}(X)$, written $\rightsquigarrow$, is defined by the axioms given in Figure 2.

As a first result, we may state the following:

PROPOSITION 1. (Grounding Lemma). *For all $t, t' \in T_{S,\Sigma}(X)$ such that $t \rightsquigarrow t'$, for all ground $\sigma'(t')$, there exists a ground $\sigma(t)$ such that $\sigma(t) \rightsquigarrow \sigma'(t')$.*

PROOF. The proof is done by structural induction over the derivation of $t \rightsquigarrow t'$, the only case which is not obvious being when the Replacement axiom is used. Suppose that $t_i \rightsquigarrow t_i'$ for $i=1, \ldots, n$ and consider some ground $\sigma'(f(t_1', \ldots, t_n'))$. By induction hypothesis, there exist some grounding substitutions σ_i such that $\sigma_i(t_i) \rightsquigarrow \sigma'(t_i')$ for all i, and, using the Replacement axiom, we obtain $f(\sigma_1(t_1), \ldots, \sigma_n(t_n)) \rightsquigarrow f(\sigma'(t_1'), \ldots, \sigma'(t_n'))$. Now as the t_is share no variables, it is possible to merge the σ_is into one single grounding substitution σ such that $\sigma(f(t_1, \ldots, t_n)) \rightsquigarrow \sigma'(f(t_1', \ldots, t_n'))$.

Transitivity

$$\frac{t_1 \rightsquigarrow t_2 \quad t_2 \rightsquigarrow t_3}{t_1 \rightsquigarrow t_3}.$$

Instantiation

$$t \rightsquigarrow \sigma(t)$$

when $\sigma \in \mathbf{WSSub}$.

Rewriting

$$\sigma(l) \rightsquigarrow \sigma(r)$$

when $\sigma \in \mathbf{WSSub}$ and $l \to r \in \mathbf{P}$.

Replacement

$$\frac{t_1 \rightsquigarrow t_1' \ldots t_n \rightsquigarrow t_n'}{f(t_1, \ldots, t_n) \rightsquigarrow f(t_1', \ldots, t_n')}$$

when $f(t_1, \ldots, t_n)$ and $f(t_1', \ldots, t_n')$ are well-sorted and when the t_is share no variables.

Figure 2. Axioms defining $\rightsquigarrow$.

† Typically, term rewriting is used in connection with equational logic, and rewrite rules $l \to r$ are obtained by orienting equations $l = r$ in which one always has $sort(r) \preceq sort(l)$, giving so-called "sort-decreasing systems" (e.g. Schmidt-Schauss (1988)). By contrast, in our formalism, we may have rules $l \to r$ with any kind of relation between $sort(l)$ and $sort(r)$. Not having this sort-decreasing property will be a major problem, but in the context of computer algebra, restricting ourselves to sort-decreasing systems would lose almost all the expressivity we need. For example, we want to be able to declare that the matrices over a commutative ring form a (not necessarily commutative) ring.

A further way in which our subtype rules differ from classical rewrite rules is that we do not require $\mathrm{Var}(r) \subseteq \mathrm{Var}(l)$. All this explains why we formulated the rewriting and replacement rules the way we did.

2.4. SEMANTICS OF TYPES

A simple set-theoretical semantics can be given to our type language. We consider a semantic domain $\mathcal{U}$ for the objects of our computer algebra language, and ground types are interpreted as subsets of $\mathcal{U}$. The meaning of polymorphic types will be defined later in term of the meaning of ground types, and in fact only ground types have a natural meaning as set of values.

Basically, the semantics we propose sticks to the "types as sets" paradigm. It is polymorphic but first-order. It also includes subtyping and inheritance, and our notion of polymorphism is refined through the use of "properties": that is, we have many-sorted (even order-sorted) polymorphism rather than just homogeneous polymorphism.

DEFINITION 7. A *model* M of the type signature $(S, \leq, \Sigma)$ is a set $\mathcal{U}$ and a semantic function $[\![\cdot]\!]$ such that:
for all $s \in S$, $[\![s]\!] \subseteq 2^{\mathcal{U}}$;
for all $s, s' \in S$, if $s \leq s'$ then $[\![s]\!] \subseteq [\![s']\!]$;
for all $f \in \Sigma$ with arity n, $[\![f]\!]$ is a partial function from $2^{\mathcal{U}^n}$ into $2^{\mathcal{U}}$;
for all $f \in \Sigma_{s_1 \times \cdots \times s_n \to s}$, the restriction of $[\![f]\!]$ on $[\![s_1]\!] \times \cdots \times [\![s_n]\!]$ is a mapping into $[\![s]\!]$;
for all $f \in \Sigma$, for all $A_i, A_i' \in [\![s_i]\!]$ such that $A_i \subseteq A_i'$, $[\![f]\!](A_1, \ldots, A_n) \subseteq [\![f]\!](A_1', \ldots, A_n')$.

This just says that a model of the type signature $(S, \leq, \Sigma)$ is an $(S, \leq, \Sigma)$-algebra where the elements are subsets of $\mathcal{U}$ and the functions preserve the subset ordering defined between these subsets of $\mathcal{U}$, as noted in section 1.2.

The semantic function defined over Σ is canonically extended over $T_{S,\Sigma}$ by recursively applying the scheme: $[\![f(t_1, \ldots, t_n)]\!] = [\![f]\!]([\![t_1]\!], \ldots, [\![t_n]\!])$.

DEFINITION 8. Given a model M of the type signature $(S, \leq, \Sigma)$, a *semantic assignment* is a mapping $\hat{\sigma}: X \to 2^{\mathcal{U}}$ such that $\forall x \in X_s$, $\hat{\sigma}(x) \in [\![s]\!]$.

Given such a mapping, it is extended canonically into an homomorphism from $T_{S,\Sigma}(X)$ to $2^{\mathcal{U}}$ and then $\hat{\sigma}(t) \in [\![s]\!]$ for all t having sort s. We write SA for the set of all semantic assignments (in a model M). Note that a grounding substitution σ automatically gives rise to a semantic assignment $[\![\cdot]\!] \circ \sigma$ but the converse is not true in general.

These assignments are used when we define what it means for a model to satisfy the subtypes rules of the presentation.

DEFINITION 9. A *model* M of the presentation $\mathbf{P} = (S, \leq, \Sigma, (l_i \to r_i)_i)$ is a model of the type signature such that, for any subtype rule $l_i \to r_i$ and any semantic assignment $\hat{\sigma} \in SA$, we have $\hat{\sigma}(l_i) \subseteq \hat{\sigma}(r_i)$.

From now on, when we speak of "a model", we always mean a model of the presentation, and not a model of the signature.

2.5. SEMANTICS OF POLYMORPHIC TYPES

The $[\![\cdot]\!]$ function may be extended in a canonical way so that it applies to polymorphic types:

DEFINITION 10. In a model M, a polymorphic type denotes the intersection of its "semantic" instances:

$$\forall t \in T_{S,\Sigma}(X), [\![t]\!] = \bigcap_{\hat{\sigma} \in SA} \hat{\sigma}(t).$$

Note that if type t of sort s is a polymorphic type, one does not necessarily have $[\![t]\!] \in [\![s]\!]$ as we do not require that sorts be preserved by set intersections. This is not a problem as a model only gives a meaning to type constructors, which combine to give a meaning to *ground types* which only then combine to give a meaning to *polymorphic types.* One should not think of these type constructors as applying to polymorphic types.†

This may seem confusing, but Proposition 5 will partially bring back the intuition one has about polymorphic types.

2.6. SEMANTICS OF SUBTYPES

In a very natural way, subtypes are just subsets:

DEFINITION 11.

Given two types $t, t' \in T_{S,\Sigma}(X)$ and a model M, we say that t is an *M-subtype* of t' iff $[\![t]\!] \subseteq [\![t']\!]$.

If t is a M-subtype of t' in every model M of **P**, we say that t is simply a *subtype* of t', written $t \lhd t'$.

We write $t \equiv t'$ when we have both $t \lhd t'$ and $t' \lhd t$.

By definition, $\lhd$ is a quasi-ordering relation (reflexivity is an instance of the Instantiation axiom), and then $\equiv$ is an equivalence relation. When we have $t \equiv t'$ for two different types, then t and t' are equivalent and indeed $[\![t]\!] = [\![t']\!]$ in every model.

With these definitions, the declarative meaning of subtype rules for ground types easily extends to polymorphic types:

PROPOSITION 2. *For any rule $l \to r$ of* **P** *and any well-sorted substitution σ, $\sigma(l) \lhd \sigma(r)$.*

PROOF. Let us consider a model M. In that model, we have $[\![\sigma(l)]\!] = \bigcap_{\hat{\rho} \in SA} \hat{\rho}(\sigma(l))$ and $[\![\sigma(r)]\!] = \bigcap_{\hat{\rho} \in SA} \hat{\rho}(\sigma(r))$. But a semantic assignment $\hat{\rho}$ and a substitution σ combine to form another semantic assignment $\hat{\theta} = \hat{\rho}.\sigma$, and for such a $\hat{\theta}$, we have $\hat{\theta}(l) \subseteq \hat{\theta}(r)$ by definition of a model. Therefore $\bigcap_{\hat{\theta}} \hat{\theta}(l) \subseteq \bigcap_{\hat{\theta}} \hat{\theta}(r)$, which is just $[\![\sigma(l)]\!] \subseteq [\![\sigma(r)]\!]$. As this is true in every model, we have $\sigma(l) \lhd \sigma(r)$.

2.7. DERIVED TYPES AND SUBTYPES

Having defined our type system, we may relate the deduction system built around the notion of derivation and the semantic notion of subtyping through the following results:

PROPOSITION 3. (Soundness)

$$\forall t, t' \in T_{S,\Sigma}(X),\ t \rightsquigarrow t' \Rightarrow t \lhd t'.$$

† This is clearer in systems (e.g. Cardelli & Wegner, 1985) where the syntax for polymorphic types explicitly includes a "for all" quantifier: what we write here t would be written $(\forall X).t$. This notation makes clear the difference between $(\forall X).FF(t)$ and $FF((\forall X).t)$: in our system, only the first one is a meaningful type. The same notation would help to distinguish between a *derivation* $(\forall X).t \rightsquigarrow (\forall Y).t'$ and a *subtype rule* $(\forall X).l \to r$.

The proof is done by structural induction over the derivation of $t \rightsquigarrow t'$. The most complicated case is when the Replacement axiom is used. To deal with this, we need a semantic equivalent of Proposition 1: *for all* $t, t' \in T_{S,\Sigma}(X)$ *such that* $t \rightsquigarrow t'$, *for any model* M, *for any assignment* $\hat{\sigma}' \in SA$, *there exists an assignment* $\hat{\sigma}$ *such that* $\hat{\sigma}(t) \subseteq \hat{\sigma}'(t')$. This lemma can be proved by structural induction on the derivation $t \rightsquigarrow t'$ and allows us to conclude. See Schnoebelen *et al.* (1988) for a complete proof.

PROPOSITION 4. (Linear Completeness)

$$\forall t, t' \in T_{S,\Sigma}(X),\ t \text{ linear},\ t \lhd t' \Rightarrow t \rightsquigarrow t'.$$

The proof is done by explicitly building a model in which, for linear t, $[\![t]\!] \subseteq [\![t']\!]$ implies $t \rightsquigarrow t'$.

We consider $T_{S,\Sigma}(X)$ itself as a semantic domain $\mathcal{U}$, and we define $\overset{-1}{\rightsquigarrow}$, from $\mathcal{U}$ to $2^{\mathcal{U}}$, by $\overset{-1}{\rightsquigarrow}(t) = \{t' \in T_{S,\Sigma}(X) \mid t' \rightsquigarrow t\}$. Note that $t \rightsquigarrow t'$ is equivalent to $\overset{-1}{\rightsquigarrow}(t) \subseteq \overset{-1}{\rightsquigarrow}(t')$. The semantics of sorts is just $[\![s]\!] = \{\overset{-1}{\rightsquigarrow}(t) \mid t \in T_s(X)\}$. For a function symbol f, we use $\overset{-1}{\rightsquigarrow}$ to define $[\![f]\!]$: suppose that $f \in \Sigma_{s_1 \times \cdots \times s_n \to s}$ and that $A_i \in [\![s_i]\!]$ for $i = 1, \ldots, n$. By definition of $[\![s_i]\!]$, there exists some $t_i \in T_{s_i}(X)$ such that $\overset{-1}{\rightsquigarrow}(t_i) = A_i$. By definition of $\overset{-1}{\rightsquigarrow}$, we have $\overset{-1}{\rightsquigarrow}(t) = \overset{-1}{\rightsquigarrow}(\theta(t))$ whenever θ is a bijective renaming of variables, therefore we may find some t_is *sharing no variables* such that for $i = 1, \ldots, n$, $\overset{-1}{\rightsquigarrow}(t_i) = A_i$. This allows us to define $[\![f]\!](A_1, \ldots, A_n)$ as $\overset{-1}{\rightsquigarrow}(f(t_1, \ldots, t_n))$.

See Schnoebelen *et al.* (1988) for a complete proof that this is a well-formed definition (i.e. our definition for $[\![f]\!](A_1, \ldots, A_n)$ does not depend on the t_is we choose for the A_is), that it does indeed give a model of the presentation, and finally that, in this model, $[\![t]\!] = \overset{-1}{\rightsquigarrow}(t)$ for linear t.

We may now relate the two notions of polymorphism and subtyping by giving the following:

PROPOSITION 5. *If* $t \rightsquigarrow t'$ *then for any ground* $\sigma'(t')$ *there exists a ground* $\sigma(t)$ *such that* $\sigma(t) \lhd \sigma'(t')$.

PROOF. Combine propositions 1 and 3.

This intuitive meaning of this proposition is that a derivation $t \rightsquigarrow t'$ can be read as $(\forall X')(\exists X).t(X) \subseteq t'(X')$.

2.8. CONCLUSION

We conclude this section by analysing the practical significance of Propositions 3 and 4. It shows that the axiomatization of $\rightsquigarrow$ correctly represents the $\lhd$ relation, but we have only proved a linear completeness result and not full completeness. However, we have been unable to disprove full completeness, that is we have not been able to find a counter-example to $\lhd \equiv \rightsquigarrow$. Indeed, we do conjecture that our definition for $\rightsquigarrow$ exhibit full completeness.

3. Semi-Decision Procedures for the Type Calculus

The type system we have just presented is very expressive, and it allows one to deal with the complex domain structure of computer algebra. We have given an adequate axiomatization for the subtype relation but we have not given a decision procedure for that axiomatization.

This section is concerned with the practical side of the problem. We shall first show that no decision procedure exists (i.e. the type inference problem is undecidable). A first solution would be to find a restricted framework in which decision procedures do exist, and where the restrictions do not give up the expressivity of the system, but we shall show that our negative answer still holds in a very simplified framework. However, we shall describe a simplified framework where it is possible to have efficient semi-decision procedures.

Throughout this section, we consider a simplified framework where there is one sort only (i.e. we do not consider properties).

3.1. A NEW DEFINITION FOR DERIVATION

In this framework, it is possible to use a simpler set of rules to define the derivation relation.

DEFINITION 12. In the simplified framework, the derivation relation is defined by keeping the Transitivity and Instantiation axioms of Definition 6 and by replacing the Rewriting and Replacement axioms by a generalized axiom. The resulting system is given in Figure 3.

This definition for $\rightsquigarrow$ has some useful consequences. First, any derivation $t \rightsquigarrow t'$ can be seen as a linear chain of deduction steps involving only the Instantiation or General Rewriting axioms. When a derivation $t \rightsquigarrow t'$ does not use the Instantiation axiom, we write it $t \overset{*}{\rightarrow} t'$ as this corresponds to the usual notion of term rewriting (modulo the fact that we do not require the variables of a right-hand side of a rule $l \rightarrow r$ to be present in the left-hand side). We shall use $\rightsquigarrow_1$ to denote the relation defined with the axioms of Figure 2 in section 2.3, and $\rightsquigarrow_2$ for the new relation. In this section, when we just use $\rightsquigarrow$ it always means $\rightsquigarrow_2$.

Our first task is to relate $\rightsquigarrow_2$ to $\rightsquigarrow_1$, which is done through the following proposition:

PROPOSITION 6.

$$\forall t, t \in T_{S,\Sigma}(X),\ t \rightsquigarrow_1 t' \Rightarrow t \rightsquigarrow_2 t'.$$

Transitivity

$$\frac{t_1 \rightsquigarrow t_2 \quad t_2 \rightsquigarrow t_3}{t_1 \rightsquigarrow t_3}$$

Instantiation

$$t \rightsquigarrow \sigma(t)$$

when $\sigma \in \mathbf{WSSub}$.

General rewriting

$$\frac{t/m = \sigma(l)}{t \rightsquigarrow t[m \leftarrow \sigma(r)]}$$

when $\sigma \in \mathbf{WSSub}$, $l \rightarrow r \in \mathbf{P}$ and m is an occurrence in t.

Figure 3. New axioms defining $\rightsquigarrow$.

We only have to check that $\rightsquigarrow_2$ satisfies the Replacement axiom of Figure 2. This is done by structural induction, with the help of judiciously chosen variable renamings. See Schnoebelen *et al.* (1988) for a complete proof.

We are now ready for the proof of adequacy between $\rightsquigarrow$ and $\lhd$:

PROPOSITION 7. (Soundness and Linear Completeness)

$$\forall t, t' \in T_{S,\Sigma}(X),\ t \rightarrow t' \Rightarrow t \lhd t',$$

$$\forall t, t' \in T_{S,\Sigma}(X),\ t \text{ linear},\ t \lhd t' \Rightarrow t \rightarrow t'.$$

PROOF. (Soundness): We just have to check that the General Rewriting axiom is sound. This stems from the fact that type constructors are interpreted as monotonic functions. See Schnoebelen *et al.* (1988) for a complete proof.

(Linear Completeness): We already know from Proposition 4 that if t is linear and if $t \lhd t'$ then $t \rightsquigarrow_1 t'$. Now Proposition 6 implies that $t \rightsquigarrow_2 t'$.

3.2. UNDECIDABILITY RESULTS

In this section, we show that there is no hope of finding a decision procedure for $\rightsquigarrow$. The statement we give is not the simplest possible, but it shows that the problem is still undecidable even when one makes strong (though reasonable) restrictions upon the subtype rules.

We suppose that:

The $\rightsquigarrow$ relation is a strict ordering (i.e. the congruence relation $\equiv$ is just syntactic equality).

The variables of the right-hand side of a subtype rule appear also in the left-hand side.

Of course, the restriction about considering only one sort still holds. We show that the problem of finding one common type (*and not necessarily all the common types*) of two given expressions is undecidable even when these restrictions are made.

PROPOSITION 8. *The problem of deciding for each pair of types t and t' if there exists a type u such that both $t \rightarrow u$ and $t' \rightarrow u$ is undecidable.*

See Schnoebelen *et al.* (1988) where the Post correspondence problem is reduced to the problem of deciding whether some t and t' have a common type.

3.3. A REFINED SEMI-DECISION PROCEDURE

The set of rules we used to define the derivation relation and the correct types of an expression immediately gives a semi-decision procedure for $\rightsquigarrow$, but the procedure is very unefficient, partly because the Instantiation rule allows any substitution to be applied. In this section we study a strategy which is much more efficient. It is complete for linear types only, which is not such a problem as $\rightsquigarrow$ itself has been proved complete w.r.t. $\lhd$ for linear types only.

This procedure is close to paramodulation (see e.g. Stickel, 1986). The basic idea is that one restricts the set of substitutions applied to a term to the substitutions which unify a subterm of the given term and a left-hand side of a rule:

DEFINITION 13. A *narrowing substitution* σ for a term t is a most general unifier of a subterm t/m of t (including the variables of t) and the left-hand side l of a subtype rule $l \rightarrow r$.

We suppose that t does not share any variable with the rule $l \to r$, which may always be achieved by renaming. The domain of σ is included in the union of the variables of t and l and we only consider idempotent unifiers, see Eder (1985).

Now if σ is a narrowing substitution for t (at occurrence m and with rule $l \to r$), then $t \leadsto t' = \sigma(t[m \leftarrow r])$ is a correct derivation. We say that "t narrows to t'" written $t \searrow t'$. This may be compared with the classical definition for narrowing (Hullot, 1980), the main difference being that reductions at an occurrence of a variable are allowed.

When studying in which cases narrowing can be used as a semi-decision procedure for type derivations, we considered the following restriction:

For any subtype rule $l \to r$, both l and r are linear

which is assumed throughout this section.† This allows a first technical lemma to be established:

PROPOSITION 9. *If t is linear and if $t \searrow t'$, then t' is linear.*

Now comes the main result of this section:

PROPOSITION 10. *For any types t and t' such that t is linear and $t \leadsto t'$, there exists a chain of narrowing derivations $t \searrow t_1 \searrow \cdots \searrow t_n$ and a substitution σ such that $t' = \sigma(t_n)$.*

PROOF. We already know that if $t \leadsto t'$, the derivation has the form of a sequence of derivation steps involving the Instantiation or the General Rewriting axioms. We show how to transform any such derivation into a narrowing derivation.

Suppose we have two consecutive derivation steps $t_1 \overset{I}{\leadsto} t_2 \overset{GR}{\leadsto} t_3$ (with t_1 linear), where $\overset{I}{\leadsto}$ and $\overset{GR}{\leadsto}$ mean that the first step involves the Instantiation axiom and the second step involves the General Rewriting axiom. Then $t_2 = \sigma(t_1)$ for some σ, and, for some occurrence m in t_2 and some rule $l \to r$, there is a substitution σ' such that $t_2/m = \sigma'(l)$, and then $t_3 = t_2[m \leftarrow \sigma'(r)]$. Now this means that $t_2/m = \sigma(t_1/m) = \sigma'(l)$, implying that t_1/m and l are unifiable. Let us write ρ for their most general unifier. Since ρ is a most general unifier of t_1/m and l, we have a narrowing derivation $t_1 \searrow \rho(t_1[m \leftarrow r]) = \rho(t_1)[m \leftarrow \rho(r)]$. As t_1 is linear, $\rho(t_1)[m \leftarrow \rho(r)]$ is linear (Proposition 9). By definition of "most general unifier", we also know that σ and σ' are specializations of ρ, which, together with the fact that $\rho(t_1)[m \leftarrow \rho(r)]$ is linear, implies that there exists a substitution σ'' such that $\sigma''(\rho(t_1)[m \leftarrow \rho(r)]) = \sigma(t_1)[m \leftarrow \sigma'(r)] = t_3$. It follows that the derivation $t_1 \overset{I}{\leadsto} t_2 \overset{GR}{\leadsto} t_3$ can be transformed into a derivation $t_1 \searrow t_2' \overset{I}{\leadsto} t_3$ where $t_3 = \sigma''(t_2')$. Note that t_2' is linear. Of course, if we just have $t_2 \overset{GR}{\leadsto} t_3$, the same result applies by considering the identity substitution $t_2 \overset{I}{\leadsto} t_2$.

Similarly, a derivation $t_1 \overset{I}{\leadsto} t_2 \overset{I}{\leadsto} t_3$ can be transformed into a derivation $t_1 \overset{I}{\leadsto} t_3$ by composing substitutions, and using these three rules, one may transform any derivation $t \leadsto t'$ where t is linear into a sequence of narrowing steps, possibly terminated by a single instantiation step.

When one analyses the proof, it appears that the restriction "the subtypes rules have linear left- and right-hand sides" is not really necessary. All that is required is that all types encountered in the derivation are linear terms. The restriction on the rules is just a way to ensure that when one starts from a linear term, only linear terms may appear during the narrowing process.

† Note that this restriction is satisfied by all the examples we gave. It is not, in practice, a limitation for types in computer algebra.

Now it is easy to see how narrowing can be used to give an efficient semi-decision procedure for type derivation. Starting from a linear type t, one just generates narrowing chains $t \searrow t_1 \searrow \cdots \searrow t_n \ldots$ and any type t' which is an instance of one of the t_is is derivable from t. If some t' is given, this procedure always stops if t' may be derived from t.

It can be shown that this procedure is not complete when non-linear terms are considered, as shown by the following example:

Let the types constructors be f, g, k, i with arity 1, and h with arity 2. Let the set of rules be

$$f(x) \to g(x) \qquad f(x) \to k(x) \qquad g(x) \to k(x)$$

and the types be $t_1 = h(f(x), f(x))$ and $t_2 = h(f(i(g(x))), f(i(k(x))))$. Then t_2 is equal to a term derived from $\sigma(t_1)$ with $\sigma(x) = i(f(x))$, but it cannot be reached by narrowing from t_1 since the operator i appears neither in the rules nor in t_1. The underlying reason is that the substitutions and the rules do not commute in the derivation relation when the linearity condition is not met, as the proof of Proposition 10 shows.

4. Typing Expressions

With the semantic framework we developed in section 2, typing expressions becomes a meaningful operation. In this section we define what is *a correct type* for an expression (given some declarations for the operators). This definition is sound in a semantic way but we do not develop this point here and remain at a less formal level of presentation. We also give precise definitions to the two notions of "most general types"and "common types" commonly used in type inference and relate them to the mechanisms used in Scratchpad II for type deduction, as they are presented in Sutor & Jenks (1987).

4.1. THE TYPING RULES

Basically, there are two ways for deducing that an expression e has type t: by using subtyping or by composing the types of its subexpressions.

Using subtyping is just using the fact that $t \lhd t'$ means that t is a subset of t'. Therefore, any expression of type t also has type t', which we may describe by the following inference rule:

$$(\mathrm{DT}) \quad \frac{x:t \quad t \to t'}{x:t'}.$$

For example, given that 3 has type *INT* and that $INT \to UP[RAT, X]$, we may deduce that 3 also has type $UP[RAT, X]$. We refer to this rule as "the *DT* rule".

The other kind of type deduction is the standard way of building types for compound expressions. Every operator in the language has a given *profile* of the form:

$$f: t_1 \times \cdots \times t_n \to t.$$

Such a declaration allows the deduction of type $\sigma(t)$ for expression $f(e_1, \ldots, e_n)$ from the n premisses $e_i : \sigma(t_i)$. This declaration can be seen as a rule:

$$(\mathrm{OP}_f) \quad \frac{x_1:t_1, \ldots, x_n:t_n}{f(x_1, \ldots, x_n):t}$$

and we call such rules "the Op rules".

Moreover, using order-sorted algebra allows us to easily express the minimal properties required from the types of the arguments of f. Since a well-formed term of sort s' is also a well-formed term of sort s for all $s' < s$, the operator is still defined if its arguments have types of a stronger sort than required.

For example, one may consider the following Op rule:

$$\frac{x{:}t{:}\mathit{AbelianGroup}\ \ y{:}t{:}\mathit{AbelianGroup}}{x+y{:}t{:}\mathit{AbelianGroup}}$$

from which, given that $2{:}\mathit{INT}{:}\mathit{CommutativeRing}$ and that

$$\mathit{CommutativeRing} < \mathit{AbelianGroup},$$

one can infer $2+2{:}\mathit{INT}$ (as the substitution "$t{:}\mathit{AbelianGroup} \mapsto \mathit{INT}$" is well sorted).

Of course, this kind of inference rules gives also the type of atomic expressions (which can be seen as nullary functions). For example one may have:

$$\frac{}{0{:}\mathit{INT}{:}\mathit{AbelianGroup}}.$$

Clearly, the DT rule is sound. The Op rules can be thought of as basic axioms giving the "type" of the function symbols used in the computer algebra system. With this, we may define what is a correct type.

DEFINITION 14. A *correct type* of an expression is any type derived for this expression according to the previous type inference rules.

That is: t is a correct type for exp iff exp:t can be derived with the DT rule and the Op rules. As an example, here is a proof tree formalizing the deduction made in section 1 (where we do not include the proof trees for derived types). It clearly shows how tedious it would be for the user to give explicitly all type information.

4.2. MOST GENERAL TYPES

Usually, an expression e may have any number of correct types. If e has type t, it also has all types derived from t. In practice, it is not easy to handle all these types, and furthermore it is not necessary to have them. Just knowing that t is a correct type is sufficient to retrieve all its derived types when they are required. This idea, already used informally at the end of section 1.1, is formalized through the notion of most general types.

DEFINITION 15. A *most general type* of an expression e is a correct type t of e such that for any other correct type t', we have $t' \rightsquigarrow t \Rightarrow t \equiv t'$. A *complete set of types* for an expression e is a set $\{t_i\}_{i\in I}$ such that every t_i is a correct type of e, and every correct type t of e may be derived from one of the t_is. A *complete set of most general types* for e is a complete set of types such that any t_i is a most general type of e.

$$\dfrac{\dfrac{\dfrac{2{:}\mathit{INT}}{2{:}\mathit{RAT}{:}\mathit{Field}}\quad\dfrac{3{:}\mathit{INT}}{3{:}\mathit{RAT}{:}\mathit{Field}}}{2/3{:}\mathit{RAT}{:}\mathit{Field}}\quad\dfrac{\dfrac{3{:}\mathit{INT}}{3{:}\mathit{UP}[\mathit{INT},X]{:}\mathit{Ring}}\quad\dfrac{\dfrac{}{X{:}\mathit{SYMBOL}}}{X{:}\mathit{UP}[t,X]{:}\mathit{Ring}}}{X{:}\mathit{UP}[\mathit{INT},X]{:}\mathit{Ring}}}{3*X{:}\mathit{UP}[\mathit{INT},X]{:}\mathit{Ring}}}{\dfrac{2/3{:}\mathit{UP}[\mathit{RAT},X]{:}\mathit{IntegralDomain}\quad 3*X{:}\mathit{UP}[\mathit{RAT},X]{:}\mathit{IntegralDomain}}{2/3+3*X{:}\mathit{UP}[\mathit{RAT},X]{:}\mathit{IntegralDomain}}}$$

Figure 4. Derivation tree for $2/3+3*X{:}\mathit{UP}[\mathit{RAT},X]$.

In general, complete sets of most general types need not exist, nor be unique, nor be finite. A special case where they always exist is when the relation $\rightsquigarrow$ is *anti-noetherian*, that is, when $\rightsquigarrow^{-1}$ is noetherian (has no infinite chains). We mention this case because, in practice, it turns out that the subtype rules we use in computer algebra often produce anti-noetherian derivation relations, and this is one more aspect where our use of rewrite rules differs from their usual applications. Note that in order to be anti-noetherian, the $\rightsquigarrow$ relation must not allow cycles, which implies that $\equiv$ is reduced to syntactic equality. It follows that when $\rightsquigarrow$ is anti-noetherian, complete sets of most general types exist and are unique.

4.3. COMMON TYPES

Suppose that we have to typecheck the expression $e_1 + e_2$ where e_1 and e_2 have types t_1 and t_2, and where the type behaviour of $+$ is given by the rule:

$$\frac{x{:}t{:}AbelianGroup \quad y{:}t{:}AbelianGroup}{x+y{:}t{:}AbelianGroup}.$$

Typing $e_1 + e_2$ requires that one finds a supertype of both t_1 and t_2, what we call a common type.

DEFINITION 16. A *common type* for two types t_1 and t_2 is a type t such that both $t_1 \rightsquigarrow t$ and $t_2 \rightsquigarrow t$ hold.

Computing common types may be seen as an instance of unification. In general, in order to apply the Op rules, we shall have to solve such problems (see below). Of course, with the notion of common types comes naturally the notions of *most general common types* and *complete sets of most general common types.*

4.4. COERCE, RESOLVE AND FORCE

The type inference facilities of Scratchpad II (Sutor & Jenks, 1987) are based on the three functions *Coerce*, *Resolve* and *Force* that we will shortly describe. We should first note that the notion of type used in Scratchpad II is slightly different (both smoother and less rigorous) from the notion we proposed in section 2: for example, the notion of property we represent by using order-sorted terms is replaced by the notion of "satisfying a predicate". As a consequence, the type system is not given a semantics w.r.t. which completeness problems could be discussed.

Coerce is used when one needs to know if objects of a given type can be coerced into another type. That is, *Coerce* is the Scratchpad II equivalent of a decision procedure for subtyping (i.e. $\rightsquigarrow$) and it may be used to (try to) decide whether a formula $t \rightsquigarrow t'$ is true or not. It is directed by several mechanisms:

If the user defines a *coerce* function for one type t to some type t', the interpreter infers that type t may be coerced to t'.

If the user defines a *map* function allowing any function from t to t' to be lifted to a function from a structured type $D(t)$ to $D(t')$, the interpreter knows that it is possible to coerce $D(t)$ to $D(t')$ if t may be coerced to t'.

In addition, the interpreter already knows "from internal system code" how to perform some specific coercions (e.g. from $UP[FF[\ldots]]$ to $FF[UP[\ldots]]$)

Resolve is used to coerce two types to a same type. That is $Resolve(t_1, t_2)$ is a type to which both t_1 and t_2 may be coerced, and this corresponds to the notion of common type from section 4.3. In Scratchpad II, *Resolve*, like *Coerce*, is driven by several mechanisms: rules and type destructuring, and there is no concern for completeness or some notion of "most general solution".

Force is used to coerce a type so that it satisfies a given predicate. For example, *INT* may be forced so that it becomes a Field when one needs a type for integers, satisfying in addition the "is a Field" property. This "forcing" is obtained by applying the *FF* type constructor and it coerces *INT* to *RAT*. This operation has an equivalent in our formalism: given a type $t:s$ and a sort s', find a derived type t' having sort s'. This may be used when we have to type an expression $f(e)$ where e has type t but where f is a polymorphic function requiring that its argument belongs to a type of sort s'. Unfortunately, this *Force* operation does not have a very regular behavior w.r.t. $\rightsquigarrow$ and there exists no notion of "most general solution" to a "*Force* $t:s$ to s'" problem. The only property one can note is that if t' is a solution, then any $\sigma(t')$ is also a solution.

These three constructs are useful when it comes to describe an actual heuristic for finding the type of a compound expression. As a typical example, suppose we have to type expression $f(e_1, e_2, e_3)$ where the e_is have type t_i. We may first look for the declared profile for f: suppose it is $f: t:s \times t:s \times t':s' \rightarrow t:s$. A possible thing to do is to find a common type t_0 for t_1 and t_2 (this is a *Resolve*) as the first two arguments of f must belong to a same type. Then, there remains to find a derived type of sort s for t_0 and a derived type of sort s' for t_3 (this is a *Force*), and we use t_0' and t_3' for these types. Now the profile for f may be instantiated by substituting t_0' and t_3' for t and t', and t_0' has been proved a correct type for $f(e_1, e_2, e_3)$.

Conclusion

In this paper, we have proposed a type system inspired by algebra. The main features are:

the use of terms of an order-sorted algebra to denote polymorphic types with properties;
the use of rewrite rules to define subtyping relations.

This work is a continuation of Comon *et al.* (1987) where we first applied our idea of using rewrite rules for expressing subtype relations between polymorphic types. Our investigation of the semantics of such a system were driven by the need to understand better what we were doing with these rewrite rules, and with that respect this work was enlightening.

Unfortunately, the type inference problem remains undecidable and we proposed only semi-decision procedures. Many research directions are still to be investigated. For example, we would like to further enhance expressivity in order to accept "polymorphic" properties instead of just a finite set of them. Another problem, and perhaps the most important from a practical point of view, is to find some stronger restrictions one could put on the formalism in order to get a decidable problem and an efficient type inference algorithm, hopefully without losing too much expressiveness.

We would like to thank Gert Smolka for the valuable comments and suggestions he made about earlier drafts of this paper, and Thierry Boy de la Tour and an anonymous referee for their thorough proof-reading and valuable suggestions which improved the paper.

References

Abdali, S. K., Cherry, G. W., Soiffer, N. (1986). An object oriented approach to algebra system design. In: *Proc. ACM Symp. Symbolic and Algebraic Computation.*

Bert, D., Echahed, R. (1986). Design and implementation of a generic, logic and functional programming language. In: *Proc. ESOP 86,* Saarbrücken, *Lecture Notes in Computer Science* **213.**

Cardelli, L. (1984). Compiling a functional language. In: *Proc. 84 ACM Conf. Lisp and Functional Programming,* Austin, Texas.

Cardelli, L., Wegner, P. (1985). On understanding types, data abstraction and polymorphism. *ACM Computing Surveys* **17(4).**

Comon, H., Lugiez, D., Schnoebelen, Ph. (1987). Type inference in computer algebra. Lecture presented at EUROCAL'87, Leipzig.

Eder, E. (1985). Properties of substitutions and unifications. *J. Symbolic Computation* **1**(1).

Fortenbacher, A., Jenks, R. D., Lucks, M., Sutor, R. S., Watt, S. M. (1985). *An Overview of the Scratchpad II Language and System.* Research Report, IBM T. J. Watson Research Center.

Futatsugi, K., Goguen, J. A., Jouannaud, J.-P., Meseguer, J. (1985). Principles of OBJ2. In: *Proc. 12th ACM Symp. Principles of Programming Languages,* New Orleans, 52–66.

Goguen, J. A., Jouannaud, J.-P., Meseguer, J. (1984). *Operational Semantics for Order-Sorted Algebra.* Research Report 84-R-101, CRIN.

Goguen, J. A., Meseguer, J. (1987*a*). Models and equality for logical programming. In: *Proc. CFLP,* Pisa, *Lecture Notes in Computer Science* **250.**

Goguen, J. A., Meseguer, J. (1987*b*). *Order-Sorted Algebra I: Partial and Overloaded Operators, Errors and Inheritance.* Draft, Computer Science Lab., SRI International.

Gordon, M. J., Milner, A. J., Wadsworth, C. P. (1979). Edinburgh LCF. A mechanised logic of computation, *Lecture Notes in Computer Science* **78.**

Hearn, A. C. (1984). *REDUCE 3.1. User's Manual.* Rand Corp., Santa Monica.

Huet, G., Oppen, D. C. (1980). Equations and rewrite rules: a survey. In: Book, R., ed. *Formal Language Theory: Perspectives and Open Problems,* pp. 349–405. London: Academic Press.

Hullot, J.-M. (1980). Canonical forms and unification. In: *Proc. 5th Conf. on Automated Deduction,* Les Arcs, *Lecture Notes in Computer Science* **87,** 318–334.

Jenks, R. D., Sutor, R. S. (1987). On the design of the Scratchpad II interpreter. Lecture presented at EUROCAL'87, Leipzig.

The Macsyma Reference Manual (version 10) (1984). M.I.T. and Symbolics Inc.

Milner, R. (1978). A theory of type polymorphism programming. *Journal of Computer and System Sciences* **17.**

Schmidt-Schauss, M. (1987). *Unification in an Order-Sorted Calculus with Declarations.* Lecture presented at Workshop on Unification, Val d'Ajoul, France.

Schmidt-Schauss, M. (1988). *Computational Aspects of an Order-Sorted Logic with Term Declarations.* PhD thesis, Univ. Kaiserslautern.

Sutor, R. S., Jenks, R. D. (1987). The type inference and coercion facilities of the Scratchpad II interpreter. In: *Proc. SIGPLAN 87 Symp. Interpreters and Interpretive Techniques,* St. Paul.

Schnoebelen, Ph., Lugiez, D., Comon, H. (1988). *A Semantics for Polymorphic Subtypes in Computer Algebra.* Research Report 711, LIFIA-IMAG, Grenoble.

Smolka, G. (1989). *Logic Programming over Polymorphically Order-Sorted Types.* PhD thesis, Univ. Kaiserslautern.

Stickel, M. (1986). An introduction to automated deduction. In: *Fundamentals of Artificial Intelligence, Lecture Notes in Computer Science* **232.**